This paper appears in the *Proceedings of the Intl. Workshop on Attribute Grammars and their Applications*, ed. by P. Deransart and M. Jourdan, Paris (F), September 19–21, 1990, pp. 29–45, vol. 461 of *Lect. Notes in Comp. Sci.*, Springer-Verlag. Copyright © 1990 by Springer-Verlag, New-York, Heidelberg, Berlin.

Space Optimization in the FNC–2 Attribute Grammar System

Catherine JULIÉ and Didier PARIGOT
INRIA*

Abstract

Memory space management for attribute evaluators is a vital requirement in practice. In fact, using attribute grammars (AGs) will very quickly meet the problem of memory space if it isn't taken into special consideration. We consider this problem for evaluators of the simple multi-visit class, also called l-ordered, because it is the largest possible AGs class for which we can find, at construction time, a method for memory space optimization. We present a new algorithm which decides, at generation time, if it is possible to store attribute instances in global stacks or global variables. The purpose of this approach is to reduce not only memory space, but also as much as possible the number of attributes to be stored in the nodes of the tree. This method is implemented in the new attribute grammar processing system, named FNC–2. Finally we present our first practical results which seem very promising.

1 Introduction

Attribute grammars (AGs) have been devised nearly 20 years ago by D.E. Knuth [1] to describe and implement the semantics of programming languages and, more generally, any syntax-directed computation. One of the main quality of this approach is that it is possible to thereby generate an attribute evaluator performing the computation specified by a given AG. Since then, much research has been done to obtain really efficient attribute evaluators using this method. A lot of research has dealt with the topic of attribute value storage, which is the main problem hindering industrial use of the AGs.

Let us briefly explain this problem. In a naive implementation, the value of an attribute instance computed during evaluation must be stored on the tree, because this value can be used for computing other attribute instances. When the last use of this value occurs for computing another attribute instance, it would be preferable to free this space for storing further values.

*Authors' address: INRIA, Bât. 52, Domaine de Voluceau, Rocquencourt, BP 105, F-78153 LE CHESNAY Cedex, France. E-mail: `{julie,parigot}@minos.inria.fr`. C. Julié is also with Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, BP 6759, F-45067 ORLÉANS Cedex 2. E-mail: `julie@univ-orleans.fr`.

We prefer the static method which considers a given attribute evaluator for an AG and determines a memory management scheme applicable to any tree. Kastens has been the first to present this kind of method for the OAG class [2], its basic idea being the notion of attribute lifetime analysis. To perform this analysis, the only required condition is that an attribute evaluation order should be statically known. And it is known that this property defines the *l*-ordered (simple multi-visit) class [3], for which attribute evaluators can be defined by visit sequences. The lifetime analysis is computed from the visit sequences.

The two most recent papers on the subject of memory space optimization, which are based on the same static method, are [7,8]. The work which is presented here has been strongly inspired by them. As a first step, let us briefly recall their methods. The basic idea is to decide if each attribute [8] or each attribute occurrence [7] can be implemented in a global variable or in a stack. The choice is done by analysing the overlappings of attribute instance lifetimes and must be valid for all the possible trees.

To store all the instances of an attribute in a variable, all their lifetimes have to be non-overlapping, and this for each tree. In the case of stack storage, they have to be properly nested or non-overlapping. Consider a given lifetime in a given production, then test if it verifies one of the two above conditions, for any tree containing this production. If for each lifetime of a given attribute the same condition has been verified, then this attribute can be implemented in a variable or a stack respectively.

To detect all the possible lifetime overlappings, the main problem is to find a mechanism for projecting the lifetimes induced by all the possible trees containing this production, onto the considered lifetime. Moreover this projection mechanism must induce necessary and sufficient conditions on the considered lifetime, to decide the mode of storage. The way to define this projection mechanism is the essential difference between the approach of [7,8] and ours (for a first presentation see [6,9]).

Kastens defines for each attribute a language which completly describes lifetime overlappings. But, as he wrote in his paper, the language is larger than the one induced by the computation sequences. We will see that it is only necessary to analyse the overlapping of the projected lifetimes on the considered lifetime and not the overlapping of these lifetimes on each other. Engelfriet and de Jong [8] have taken this problem into consideration, but as we will show, they still compute more information than necessary.

Our projection method is based on the construction of pseudo context-free grammars according to the structure of the visit-sequences which compose the given evaluator. These pseudo-grammars allow to compute the projected information but they don't explicitely contain them. They just represent the visit sequence overlapping, independently of the attributes (which is the first advantage of this method). For a given attribute, we will use the pseudo-grammars in addition to an information on the lifetimes of this particular attribute. In fact it is possible to define this information in a minimal way, and this seems easier than the formalism of [8].

Moreover, in the case of temporary attributes, we consider unstrict stacks, i.e., we allow to access not only the top element, but also a bounded number of elements below it [6,9]. This is very interesting because it allows to store all the temporary attributes in such stacks. Note that the problem of their management is a purely technical implementation one. When storing all the temporary attributes in stacks, it must be clear that we seek not only to optimize memory space, but also to avoide storing values in the tree.

Secondly, it is desirable to group these objects in order to reduce memory space and remove a maximum number of copy rules: if two attribute instances are stored in the same memory space, then it is possible to eliminate copy rules between them. We propose an algorithm for regrouping variables and stacks which gives priority to the elimination of copy rules.

Our main purpose being the practical results, we have implemented our method in a new attribute grammar processing system, named FNC-2 [6,10]. A large part of this system has been specified by AGs, and the results of its own generation (bootstrap) are very promising concerning memory space (see section 7). Because of the concrete realization purpose, our approach differs from the other ones: we have been forced to distinguish the cases of temporary and non-temporary attributes, even if both cases can theoretically be unified as in [8]. According to numerous experiments [6,9,11,12] it has been shown that temporary attributes represent more than 90% of attributes. Moreover it has been proved that every temporary attribute can be stored in a stack, in the worst case. On the other hand, we will see that for non-temporary attributes we need more informations, but these informations can be computed from those of temporary attributes. For all these reasons, we have first implemented the method for temporary attributes only.

The paper is organized as follows. In section 2, we introduce the notations and definitions which will be used in the following sections. Section 3 is an overview of the space management problem and defines the purpose of our work, i.e., computing the minimal information to be projected. In sections 4 and 5, we deal with temporary attributes and non-temporary ones, respectively. In section 6 we present a new algorithm for grouping together variables or stacks. In section 7, we present practical results obtained by running the FNC-2 system. In section 8 we give some conclusions on this work and also sketch future works.

2 Definitions

In this section we introduce the preliminaries of our terminology and notations, but we assume that the reader has a basic understanding of attribute grammars. The reader is referred to the literature on this topic, for instance the survey [13] of AGs. But we will always refer to the same example, which is a good introduction to our notations.

An AG is based on a context-free grammar $G = \{N, T, P, Z\}$ consisting of non-terminals, terminals, production rules and initial non-terminal respectively. It is augmented by a set of attributes A and a set of semantic rules which define the computation of attribute values.

- Each non-terminal $X \in N$ has two associated disjoint, finite sets, denoted $IA(X)$ and $SA(X)$, of inherited and synthesized attributes, respectively. When considering a production rule of the form $X_0 \longrightarrow X_1 \ldots X_n$, we denote an attribute a of X_j ($j \in [0..n]$) as (a, j).

- Each production $p \in P$ has a set of semantic rules that define the attributes of $SA(X_0)$ and of $IA(X_j)$ for $j \in [1..n]$, which we call the output attributes of p. Each of these attributes is defined by a function depending on some attributes of $IA(X_0)$ and $SA(X_j)$ for $j \in [1..n]$, which we call the input attributes of p. We denote by $D(p)$ the local dependencies of p.

We will only consider evaluators for the subclass of the AGs called l-ordered [4]. An AG is l-ordered iff there exists, for each non-terminal X, a totally ordered partition of its attributes $\pi(X) = IA_1(X),\ SA_1(X),\ \ldots IA_{last(X)}(X),\ SA_{last(X)}(X)$ such that, for each production p, the graph, noted $T(p)$, obtained by pasting together the local dependencies $D(p)$ and the dependencies between the attributes of each non-terminal X appearing in p induced by $\pi(X)$ is acyclic.

Then in every tree t, there exists a total evaluation order, denoted $T(t)$, which can be obtained by combining the orders $T(p)$ for each production p appearing in t. During evaluation, the inherited attributes of $IA_i(X)$ are evaluated before the i-th visit to a node labelled by X, and the synthesized attributes of $SA_i(X)$ are evaluated during this i-th visit. We denote by $last(X)$ the last visit to a node labelled with X.

In what follows, assume that G is an l-ordered AG. For any production $p : X_0 \longrightarrow X_1 \ldots X_n$ of G, a visit-sequence for p (satisfying the l-ordered definition), denoted by O_p, consists of k parts $O_p(1), \ldots, O_p(last(X_0))$, where each piece $O_p(i)$, for $i \in [1..last(X_0)]$), has the form :

$$\begin{aligned} O_p(i) = {} & \texttt{EVAL}\ (IA_{j_1}(X_{l_1})),\ \texttt{VISIT}\ (X_{l_1}, j_1), \ldots, \\ & \texttt{EVAL}\ (IA_{j_q}(X_{l_q})),\ \texttt{VISIT}\ (X_{l_q}, j_q),\ \texttt{EVAL}\ (SA_i(X_0)),\ \texttt{LEAVE}(X_0, i) \end{aligned}$$

The visit-sequences are fixed at evaluator generation time, on the basis of the attribute dependencies specified in the AG. They determine an evaluator for the l-ordered AG, denoted SMV-evaluator, which functions as follows. Let t be a structure tree for a sentence of $L(G)$. In order to compute a value for all attribute instances in the tree t, the evaluator walks through it. Visiting a node n labelled with a production p, it will perform three kinds of operations :

- `EVAL` (A_k) with $A_k = IA_k(X_j)$ for $j \in [1..n]$ and $k \in [1..last(X_j)]$ or $A_k = SA_k(X_0)$ for $k \in [1..last(X_0)]$. Such an `EVAL` instruction corresponds to the evaluation of some output attributes of p.
- `VISIT` (X_j, i) with $i \in [1..last(X_j)]$ corresponds to the i-th visit to the j-th successor node of n.
- `LEAVE`(X_0, i) with $i \in [1..last(X_0)]$ means to leave n for its parent for the i-th time.

Let us give three more specialized notions for our lifetime analysis.

Definition 2.1 (Usage-section and Lifetime-section) *Let (a,i) be an input attribute of a production $p : X_0 \longrightarrow X_1 \ldots X_n$. An* `EVAL` *instruction uses (a,i) when the* `EVAL` *instruction calls for the evaluation a semantic rule using (a,i).*

- *Let (a,i) be an inherited attribute, i.e. $i = 0$. Assume that $a \in IA_j(X_0)$. A usage-section of $(a,0)$ in p, is any subsequence of O_p that begins with the beginning of $O_p(j)$ (A_f) and ends in an* `EVAL` *instruction of O_p that uses $(a,0)$ (A_l). The lifetime-section of $(a,0)$ in p, denoted $L = A_f\ B\ A_l$, is its longest usage-section.*
- *Let (a,i) be a synthesized attribute, i.e. $i \in [1..n]$. Assume that $a \in SA_j(X_i)$. A usage-section of (a,i) in p is any subsequence of O_p that starts with the* `VISIT` (X_i, j) *instruction (A_f), and ends with an* `EVAL` *instruction of O_p that uses (a,i) (A_l). The lifetime-section of (a,i) in p, denoted $L = A_f\ B\ A_l$, is its longest usage-section.*

Notice that a lifetime-section could be a subsequence of $O_p(j)$ but could also span across several parts of O_p (i.e. $O_p(j) \dots O_p(j+l)$).

Definition 2.2 (temporary attribute) *An attribute $a \in A(X)$ is temporary if the following is true. Inherited and synthesized attributes are considered separately.*

- *Let a be an inherited attribute. Assume that $a \in IA_j(X)$. Then, for every production p having X at the left hand side (LHS), the last* `EVAL` *instruction of O_p using $(a, 0)$ is inside $O_p(j)$.*
- *Let a be a synthesized attribute. Assume that $a \in SA_j(X)$. Then, for every production $p : X_0 \longrightarrow X_1 \dots X_n$ such that $X_i = X$ ($i \in [1..n]$), if* `VISIT` *(X_i, j) occurs in $O_p(l)$, then the last* `EVAL` *instruction of O_p using (a, i) is also contained in $O_p(l)$.*

This ends the definitions we need to describe the tests that decide whether an attribute can be stored in a global variable or in a stack.

3 Preliminaries

Our purpose is to define a transformed SMV-evaluator that stores attributes in a space-saving manner. More precisely, it can store some attributes in global variables and some in stacks. The transformed SMV-evaluator stores all the instances of an attribute (a, X) (simply noted a in the following) in a global variable, called $V[a]$, if, whenever it computes an instance of a, it stores this value in $V[a]$, and, whenever it needs the value of an instance of a, it uses the value of $V[a]$. Similarly, the transformed SMV-evaluator stores an attribute a in a stack, called $S[a]$, if, whenever it computes the value of an instance of a, it pushes this value onto $S[a]$. Moreover, whenever the evaluator needs the value of an instance of a, it uses the top of $S[a]$ (or k (bounded) elements below the top of $S[a]$ in the case of temporary attributes). Clearly, it will also perform some pop operations on $S[a]$ when the top values have become useless. In what follows $E(V, S)$ indicates a SMV-evaluator that stores the attributes of V in global variables, and those of S in stacks. Naturally we want to be sure that V and S are such that $E(V, S)$ behaves as E. $E(V, S)$ is correct if for every complete tree t of G, the semantic value of t computed by $E(V, S)$ is equal to the semantic value computed by E. Given a tree t of G, we define the computation sequence, denoted $CS(t)$, as the combination of the visit-sequences associated to productions which appear in the structure of t. Using such a computation sequence, it is not difficult to state two conditions that guarantee the correctness of $E(V, S)$.

Condition 3.1 (on V) *For each a in V, for each complete tree t of G, and for any two instances of a in t, their lifetimes in $CS(t)$ are strictly disjointed.*

Condition 3.2 (on S) *For each a in S, for each complete t of G, and for any two instances of a in t, their lifetimes in $CS(t)$ are either disjointed or properly nested.*

In what follows, tests for conditions (3.1) and (3.2) will be described. The intuitive idea behind these tests is that the lifetime of an attribute inside a computation sequence $CS(t)$ corresponds to the lifetime inside some visit-sequence O_p.

Consider, for instance, the lifetime-section $L = A_f \ B \ A_l$ in O_p of an attribute a. We want to determine if the instruction sequence B can produce lifetimes of a, for each complete tree t containing this production p. There are two main cases:

1. The generation by the instructions sequence B depends only on the p production (i.e. the lifetimes produced correspond to output occurrences of a in p). In this case, it is easy to perform the tests of our conditions (3.1) and (3.2) locally.

2. The generation by the instructions sequence B depends on a tree structure which can be constructed with this p production. The instruction sequence B can produce another lifetime coming from another production, through the `VISIT` or `LEAVE` instructions in B. In this case, the main problem is to define correctly the lifetime projection of B instructions on L, for each complete tree containing the p production.

Let us give some explanation about the definition of this lifetime projection. The knowledge of the relative order between the lifetimes projected through B isn't necessary. One only needs to check whether these lifetimes end inside B or overlap L. In fact, the correctness of the order will be verified locally, during the test of their corresponding lifetime-sections. In the same way, the longest lifetime (i.e. the worst case of all the possible trees) is sufficient to test if this ends really inside B.

Assume that B in O_p doesn't contain any `LEAVE` instructions. Then, the projection of lifetimes depends only on all the subtrees of the production p. According to the definition of temporary attributes 2.2, we just need a bottom-up projection mechanism induced by the `VISIT` instructions. Assume now that B in O_p contains at least one `LEAVE` instruction. Then the attribute is not temporary, and we will need a top-down projection mechanism (induced by the `LEAVE` instructions) in addition to the bottom-up one.

So we begin the presentation of our method by considering first the case of temporary attributes, and then the non-temporary one. That seems to be a good decomposition of the problem because the analysis of temporary attributes is much easier, and the other one is in fact a natural extension of the first case.

4 Space Management for Temporary Attributes

In this section, tests will be described that check precisely whether a temporary attribute ($a \in A(X)$) can be stored in a global variable or in a stack.

4.1 Global Variables

Given the lifetime-section of an input occurrence of a in a production p, we have to state whether a new occurrence of a will be defined inside this lifetime-section. As we have seen below, we only need a bottom-up projection mechanism. This projection will be computed with the Grammar of Visits, defined as follows.

Definition 4.1 (Grammar of Visits G_V) *G_V is a context-free grammar $(N_V, \emptyset, P_V, Z_V)$. N_V contains all the symbols of the form (X, i), where X is a non-terminal of G and*

$i \in [1..last(X)]$. The initial non-terminal is $Z_V = (Z, 1)$. P_V is defined as follows: For each production $p : X_0 \longrightarrow X_1 \dots X_n$, and for each $j \in [1..last(X_0)]$ let $O_p(j)$ be the j-th visit-sequence of p. P_V contains the production $p_j : (X_0, j) \longrightarrow (X_1, k_1) \dots (X_m, k_m)$ where the (X_i, k_i) are such that there exists an instruction `VISIT` *(X_i, k_i) in $O_p(j)$.*

Let G_V be the Grammar of Visits of G. We express the "syntactic dependencies" between the non-terminals of G_V defined by the relation $\underset{G_V}{\Longrightarrow}$. Then, for each $(Y, j) \in N_V$, $(Y, j) \underset{G_V}{\overset{*}{\Longrightarrow}} (X, k)$ iff an occurrence of a could be defined during the j-th visit of a node labelled with Y.

Test 4.1 (temporary attribute stored in a global variable) *A temporary attribute $a \in A_k(X)$ can be stored in a global variable iff conditions (i) and (ii) are satisfied:*

i) For each production $p : X_0 \longrightarrow X_1 \dots X_n$ and for every two input occurrences (a, i) and (a, j) of a in p, the lifetime-sections of (a, i) and (a, j) are disjointed.

ii) For each production $p : X_0 \longrightarrow X_1 \dots X_n$ and for any input occurrence (a, i) of $a \in p$, if the sequence of `VISIT` *instructions of the lifetime-section of (a, i) in p is* `VISIT` *$(X_{l_1}, j_1) \dots$* `VISIT` *(X_{l_q}, j_q) then, for every $h \in [1..q]$ it is* **not** *the case that in G_V $(X_{l_h}, j_h) \underset{G_V}{\overset{*}{\Longrightarrow}} (X, k)$.*

4.2 Stacks

We give first the test for checking whether a temporary attribute can be stored in a stack or not. Remember that, for practical reasons, we allow to access not only the top of the stack, but also to elements inside the stack.

Test 4.2 (temporary attribute stored in a stack) *A temporary attribute $a \in A_k(X)$ can be stored in a stack iff the following condition is satisfied. For each production $p \in P$ and for any two input occurrences (a, i) and (a, j) of a in p, if the lifetime-section of (a, i) in p, denoted L_i, contains an element of the lifetime-section of (a, j) in p, denoted L_j, then L_i must contain the whole lifetime-section L_j.*

Notice that the test is completely local for a temporary attribute stored in a stack: we are just interested in checking whether the lifetimes of two occurrences of a in a production are non-overlapping or properly nested. But we don't need any information on the Grammar of Visits (as it is the case for the global variable test). The reason is that if, inside the studied lifetime-section, a new occurrence of a is defined during the visit of a subtree of a node labelled by p, then its lifetime is completely contained in this visit (see the definition (2.2) of temporary attributes). Thus, those two lifetimes are properly nested.

Assume now there exists an overlap between two lifetime-sections, denoted L_i and L_j. Then, test 4.2 will fail and thus the attribute can't be implemented in a stack. But it is possible to transform the lifetime-sections so that they become properly nested. Intuitively, the method consists in extending one of the lifetimes: this extension leads to delay the pop instruction. In the worst case, one can keep the value in the stack until the end of the visit, instead of popping this value when it becomes useless.

Other differences between our work and others are the use of unstrict stacks and the transformation. Together, they allow to store all the temporary attributes in stacks.

5 Space Management for Non-Temporary Attributes

In what follows we will consider non-temporary attributes. Assume that a is a non-temporary attribute of G. Then, there exists an input occurrence (a,i) of a in some production p as usual, such that its lifetime is not completely contained inside one visit to X_0. In other words, there exists a lifetime that spans across several visits to a node labelled by X_0.

5.1 Global Variables

The lifetime analysis is similar to that for temporary attributes: one must check whether the lifetimes are non-overlapping or not. But to generate the top-down projection mechanism, we need to complete the Grammar of Visits with the different visits that could be performed between two consecutive visits to a given non-terminal (i.e. by a `LEAVE` instruction). This is done by the Grammar of Contexts and Visits which is defined as follows.

Definition 5.1 (Grammar of Contexts and Visits G_{CV}) *G_{CV} is a context-free grammar*
$(N_{CV}, \emptyset, P_{CV}, Z_V)$.

> *i) N_{CV} contains all non-terminals of N_V and, for each non-terminal $X \in N$ such that $last(X) \neq 1$ and for each $i \in [1..last(X)-1]$, a symbol denoted $(\widetilde{X}, i)$.*
>
> *ii) P_{CV} contains all the productions of P_V and also productions described as follows.*
>
> *For each $X \in N$ such that $last(X) \neq 1$ and for each $j \in [1..last(X)-1]$, consider each visit-sequence O_p that contains the instructions* `VISIT` (X,j) *and* `VISIT` $(X,j{+}1)$*. For each subsequence of O_p having the form:* $\ldots$`VISIT` $(X_q,j)\; w_1 \ldots w_m$ `VISIT` $(X_q,j$
> $1)\ldots$*, P_{CV} contains the production $\widetilde{p} : (\widetilde{X_q}, j) \longrightarrow \alpha_1 \ldots \alpha_m$ where for each $l \in [1..m]$*
>
> $\alpha_l = (X_i,k)$ *iff* $w_l =$ `VISIT` (X_i,k) *or*
> $\alpha_l = (\widetilde{X_0},k)$ *iff* $w_l =$ `LEAVE` (X_0,k) *or*
> $\alpha_l = \epsilon$ *iff* $w_l =$ `EVAL` (A).

Intuitively, a production $\widetilde{p} : (\widetilde{X}, j) \longrightarrow \alpha_1 \ldots \alpha_m$ describes the potential visits to the above productions and their subtrees, that could happen between the j-th and the $j+1$-th visit to a node labelled by X.

Let G_{CV} be the Grammar of Contexts and Visits for G. Then, for each $(\widetilde{X}, i) \in N_{CV}$, $(\widetilde{X}, i) \underset{G_{CV}}{\overset{*}{\Longrightarrow}} (Y,j)$ iff the j-th visit to a node labelled by Y could be performed between the i-th and $i+1$-th visit to a node labelled by X.

Test 5.1 (non-temporary attribute stored in a global variable) *A non-temporary attribute*
$a \in A_k(X)$ can be stored in a global variable iff conditions (i) and (ii) below are satisfied:

i) For each production $p : X_0 \longrightarrow X_1 \ldots X_n$ and for every two input occurrences (a, i) and (a, j) of a in p, the lifetime-sections of (a, i) and (a, j) are disjointed.

ii) For each production $p : X_0 \longrightarrow X_1 \ldots X_n$ and for any input occurrence (a, i) of a in p, let $L_a = A_f\ \beta_1 \ldots \beta_m\ A_l$ be the lifetime-section of (a, i) in O_p. Then, for $l \in [1..m]$ it is **not** *the case that*

$$\beta_l = \mathtt{EVAL}\ (A_k(X)) \quad or$$
$$\beta_l = \mathtt{VISIT}\ (Y, j) \ \ and \ \ (Y, j) \overset{*}{\underset{G_V}{\Longrightarrow}} (X, k) \quad or$$
$$\beta_l = \mathtt{LEAVE}\ (Y, j) \ \ and \ \ (\widetilde{Y}, j) \overset{*}{\underset{G_{CV}}{\Longrightarrow}} (X, k).$$

5.2 Strict Stacks

For each attribute we define a set of triplets (X, r, s), called Lifetime-Set. Intuitively, $LSET(a)$ contains an element of the form (X, r, s) iff there exists an occurrence of a defined during the r-th visit to a node labelled by X which will be used for the last time during the s-th visit to this node. Notice that it is possible that $r = s$.

Definition 5.2 (Lifetime-Set $LSET(a)$) *For each production $p : X_0 \longrightarrow X_1 \ldots X_n$ and each outside occurrence (a, i) of a in p let $L = \beta_1 \ldots \beta_l$ be the lifetime section of a in p, such that $\beta_1 \in O_p(r)$ and $\beta_l \in O_p(s)$, $r \leq s$. Then $LSET(a)$ contains the triplet (X_0, r, s).*

Definition 5.3 (Grammar of Lifetimes G_L) $G_L = (N_L, \emptyset, P_L, (Z, 1, 1))$ *where*

- *N_L contains all symbols of the form (X, i, j) and $(\widetilde{X}, i, j)$ where X is a non terminal of G and $1 \leq i \leq j \leq last(X)$.*
- *The productions of P_L correspond to upward and downward projections defined below.*

  i) For each $(X, i, j) \in N_L$ and each production $p : X_0 \longrightarrow X_1 \ldots X_n$ such that $X_k = X$ $(k \in 1 \ldots n)$, if $O_p(r)$ is the component of O_p that contains VISIT(X_k, i) and $O_p(s)$ $(r \leq s)$ that that contains VISIT(X_k, j) then P_L contains the production $(X_0, r, s) \longrightarrow (X_k, i, j)$.

  ii) For each $(X, i, j) \in N_L$ and for each production $p : X_0 \longrightarrow X_1 \ldots X_n$ such that $X_k = X$ $(k \in 1 \ldots n)$, consider in O_p the subsequence w that begins with VISIT(X_k, i) and ends with VISIT(X_k, j). For each $l \in 1 \ldots n$ with $l \neq k$, consider the visit-instructions concerning X_l present in w. Let all of them be VISIT$(X_l, r) \ldots$ VISIT(X_l, s) for $r \leq s$. In this case, P_L contains the production $(\widetilde{X_l}, r, s) \longrightarrow (X_k, i, j)$.

  iii) For each $(\widetilde{X}, i, j) \in N_L$ and each production $p : X_0 \longrightarrow X_1 \ldots X_n$ such that $X = X_0$, consider $w = O_p(i) \ldots O_p(j)$. For each $l \in 1 \ldots n$ such that there is at least one VISIT concerning $X_l inw$, let all of them be VISIT$(X_l, r) \ldots$ VISIT(X_l, s) for $r \leq s$. In this case, P_L contains the production $(\widetilde{X_l}, r, s) \longrightarrow (\widetilde{X}, i, j)$.

Test 5.2 (non-temporary attribute stored in a stack) *A non-temporary attribute $a \in A(X)$ can be stored in a stack iff the two following conditions are satisfied.*

i) The same as for temporary attributes.

ii) For each production $p : X_0 \longrightarrow X_1 \dots X_n$ and any outside occurrence (a, i) of a in p, let $US = \beta_1 \dots \beta_l$ be a usage-section of a in p such that $\beta_1 \in O_p(r)$ and $\beta_l \in O_p(s)$ $(r \leq s)$. The following two points must be satisfied:

- *There must not be a non-terminal $(\widetilde{X_0}, q, m)$ such that $(\widetilde{X_0}, q, m) \Longrightarrow (X', q', m')$ with $(X', q', m') \in LSET(a)$, and $r < q \leq s \leq m$.*
- *There must not be a non-terminal (X_k, q, m) $(k \in 1 \dots n)$ such that $(X_k, q, m) \Longrightarrow (X', q', m')$ with $(X', q', m') \in LSET(a)$, and for which $VISIT(X_k, q)$ is contained in US but $VISIT(X_k, m)$ is not.*

6 Grouping together Stacks and Variables

Our purpose is to define an algorithm for grouping together stacks and variables, i.e. combining several global variables (stacks) into a single one. This policy not only reduces memory space still more, but can also produce some startlingly efficient optimizations if it is done in order to remove as many copy rules as possible. It must be clear that such semantic rules don't create any new information, and removing them has a beneficial effect on run-time evaluation.

All copy rules between two given attributes can be eliminated if both attributes are sharing the same space. But we apply the grouping algorithm once we have determined which attributes are to be implemented as global variables or stacks. The reason is that we don't want to reconsider the decision of the previous tests, because using variables indisputably provides more important storage optimizations. Moreover, storage in variables doesn't influence run time while it is the case for storage in stacks, because of the push and pop instructions. Our grouping method is done in two steps.

First, we build a graph of incompatibilities, denoted IG, to memorize the attributes that can't share the same space. More precisely, the vertices of IG correspond to the attributes names, and there exists an edge between two vertices a and b, a and b with both implemented in variables (respectively in stacks), iff it isn't possible to combine them into a single variable (respectively stack). The IG graph is built by applying the previous conditions, except we generalize the analysis to several attributes. For instance, assume that a and b are two attributes that can be implemented in global variables. Let L_a be the lifetime-section of an occurrence of a in a given production p. Then we add an edge between a and b in the IG graph iff a lifetime-section of an occurrence of b begins inside L_a.

Secondly, once the IG graph has been built, the problem of grouping together variables and stacks is a vertices coloration problem. For our purpose, colors correspond to the names of variables and stacks. Giving two vertices a and b such that there exists no edge between them, it is clear that the two corresponding attributes can be implemented into a single space: thus, we need only one variable (or one stack) to store all the values of a and b.

```
build $IG$ graph;
**repeat**
  $visit\_X[Z,1] := true$;
  **for each** $X$ in $N$ **do**
    **for** $j := 1$ **to** $last(X)$
      **if** $visit\_X[X,j]$ **then begin**
        **for each** $Op(j)$ that $LHS(p) = X$ **do**
          **for each** instruction INST in $O_p(j)$ in sequence **do**
            **case** INST **is**
              EVAL $(A_k(X_i))$ : $color(A_k, IG)$;
              VISIT $(X_i, k)$ : $visit\_X[X_i, k] := true$;
            **end case**
        $visit\_X[X,j] := false$; { (X,j) processed}
      **end**
**until** all the attributes are colored
```

Figure 1: The Grouping Algorithm of Variables or Stacks

It has been proved that this problem of coloration is NP-complete if one is looking for an optimal scheme. So, we propose a non-optimal algorithm for coloring the IG graph, that is for determining the final allocation scheme for the attributes.

In fact, we have to define an order for coloring the vertices. In our context, it seems natural to choose an order induced by the visit-sequences, according to the structure of the evaluator. Such a coloration order will be define by walking through the evaluator (see figure 1). Let us give the basic idea of the algorithm.

While not all the attributes are colored, let us consider a visit-sequence $O_p(i)$ such that $X = LHS(p)$ and `VISIT` (X,i) has been previously examined. We treat the instructions of $O_p(i)$ in the order they appear in the visit-sequence. Consider now an instruction of $O_p(i)$.

If it is an `EVAL` (A_k) instruction, then we color as soon as possible the attributes of A_k, depending on the colors of those which appear in the right side of the corresponding semantic rules. Of course, if an attribute of A_k depends on nothing it will be colored at the beginning of the algorithm. Note it is the case for the inherited attributes of the initial non-terminal, and for the pure synthesized attributes.

If the instruction is a `VISIT` (X_j, k) one, then we memorize that the k-th visit of X_j has been treated. At the beginning, we need to mark the first visit of the initial non-terminal.

During the step that colors some attributes of a given partition A_k, it is possible to add some priorities, so that we first color the attributes defined by a copy rule. Doing that, we will see in the following section that the percentage of eliminated copy rules is quite important. Moreover, we will show that our algorithm remains non-optimal for only 12 % of the copy rules (see table 1).

7 Implementation and Practical Results

In this section we present our practical results obtained through the bootstrap of our system, FNC-2 [6]. We only have implemented the space management of temporary attributes, but it must be clear that to implement the second part (the space management for non-temporary attributes), we just need to build our grammar of contexts and visits. The first version of the FNC-2 system was born one year ago, and it is largely specified by AGs written in OLGA (a new AG-description devoted language, see [10]). And so, we present practical results concerning some of those AGs (AG1 to AG7 in table 1).

For each AG we will use the following notations, to show their sizes and the results of our optimization method:

- nba : number of attributes;
- nbnt : number of non-terminals;
- nbprod : number of productions;
- nbrules : number of semantic rules;
- atr$_{ave}$: average number of attribute occurrences per non-terminal;
- variables : percentage of attribute occurrences stored in variables;
- stacks : percentage of attribute occurrences stored in stacks;
- tree : percentage of attribute occurrences stored in tree (non-temporary attributes);
- copy/rules : percentage of copy rules removed, w.r.t the total number of semantic rules;
- copy/copy: percentage of copy rule removed, w.r.t the total number of copy rules;
- copy/copy$_T$: percentage of copy rules removed. w.r.t the number of copy rules between two attributes in the same type of storage space(i.e. variable or stack);
- copy/copy$_{IG}$: percentage of copy rules removed, w.r.t the number of copy rules between two attributes in the same type of storage space that are compatible (i.e no edge between them in the IG graph);

The averages of each result are weighted by the number of attribute occurrences of each AG.

First, table 1 shows that more than 92% of attributes are temporary, and thus implemented as variables or stacks ($\simeq$56.5% for variables and 36% for stacks). Moreover, the Grammar of Visits allows to store more than 50% of attributes in global variables. Only 7.5% of attributes are not temporary, and this shows that the practical interest of the second part of our work is rather low. In fact, the implementation of the method for non-temporary attributes seems interesting more in the sense that it allows to remove all the attribute values from the tree, rather than to minimize memory space.

As for the elimination of copy rules, notice that the last percentage, called copy/copy$_{IG}$, is close to 88 %: this number indicates that 12 % of the copy rules remain which could have been removed in the best case, by using a NP-complete coloration algorithm. In others words, our coloration algorithm is non-optimal for only 12 % of the copy rules.

In figure 2, we are interested in the gain of memory space during runtime. This graph shows for two AGs on various kinds of source texts, the ratio between the number of cells used to store attribute values, for a non-optimized evaluator and the corresponding optimized one. This ratio asymptotically tends towards values between 4 and 8, depending

	attributes grammars							weighted average
	AG1	AG2	AG3	AG4	AG5	AG6	AG7	
nba	30	77	16	7	103	7	30	
nbnt	12	35	35	6	74	74	74	
nbpro	47	131	131	27	319	319	319	
atr_{ave}	8.17	9.14	2.69	3.33	13.4	0.97	3.36	
nbrules	417	1420	444	99	5322	451	1497	
variable	86	61	33	80	60	48	35	56.5
stacks	7	35	62	20	29	52	64	36
tree	7	5	5	0	11	0	1	7.5
copy/rules	45	16	15	42	22	10	23	22
copy/copy	64	22	33	88	27	28	36	30
copy/copy	68	25	34	88	33	28	38	34
$\mathrm{copy/copy}_T$	75	46	46	88	51	52	55	52
$\mathrm{copy/copy}_{IG}$	88	84	80	100	91	80	88	88

Table 1: Static Results of Optimized Evaluators on our AGs

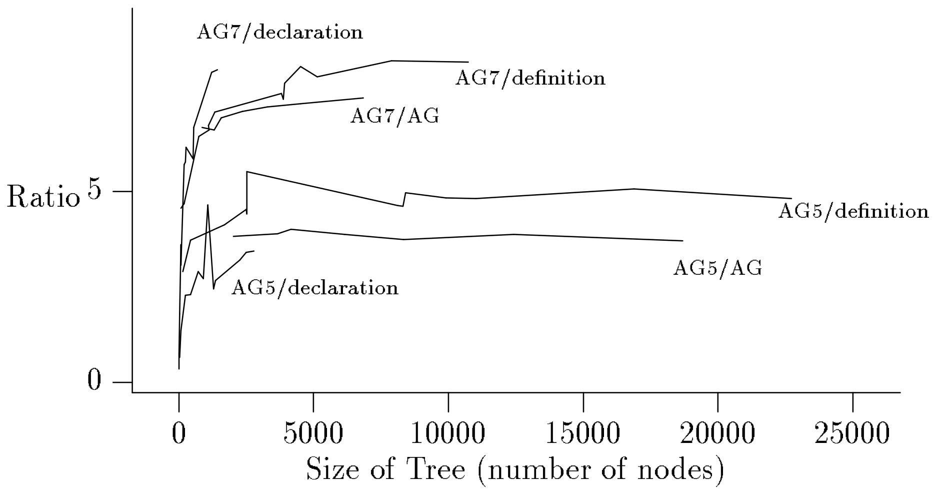

Figure 2: Dynamic Results of Optimized Evaluators on our AGs

on the evaluator and the type of the input text. This meens we use 4 to 8 times less space, with optimized evaluators.

It is difficult to give comparable results for the other methods [7,8,12] since we cannot use the same AGs with the same description language and the same order construction algorithm. More precisely, to have a correct comparison, we must use the same visit sequences as input. Secondly, we have given a method which describes necessary and sufficient conditions. In consequence, given a total order of evaluation, our results are optimal. Our results are thus a characteristic of the AG. Engelfriet and de Jong do not give any practical results [8]. Consequently, we can only compare theoretically. Farrow and Yellin [12] deal with an alltogether different aspect of the problem: choosing the total order for improving space storage optimization. In some cases, our order derives from the transformation (strongly noncircular to l-ordered) [6]. Our first criteria for choosing the order, is reducing the size of the resulting l-ordered AG.

8 Conclusion and Future Works

We have presented in this paper a method based on static attribute lifetime analysis to determine whether attributes can be implemented either in global variables or in stacks.

Remember that our main goal was to store all the attributes outside the tree. Here, we have restricted the test for non-temporary attributes to strict stacks (access only to the top), but we are sure that it is possible to remove this restriction (as we have done for temporary attributes) by using special stacks (cactus-stacks). The management of such stacks needs much more explanation than is the case for temporary attributes, and it would be too long here to expose in detail the definition and the use of such a cactus-stack, and the algorithm that modifies consequently the visit-sequences.

While for temporary attributes, the values associated to a given production are always adjoining in a stack, it is not true for non-temporary attributes. So, we must take special care in the way we access the elements. To manage correctly these accesses, the idea consists in decomposing the stack in several parts linked together. This defines a new structure called cactus-stack which is isomorphic to the computation sequences. The cactus-stack management can be done by adding in the visit-sequences some appropriate instructions, which allow to build and also to walk through this structure.

Giving such cactus-stacks, it is then possible to store all the attributes in them (i.e., outside the tree). Intuitively, if there exists an overlapping of lifetimes, it could be eliminated. The transformation will be done, in the worst case, by adding a visit to correctly delay the pop instruction.

We are going to implement that space management method for non-temporary attributes, to test if such a technic is realistic for practical use. But the most attractive consequence of removing all the attributes from the tree is that it allows a greater power of expression for attribute grammars. In particular, given some restrictions, the evaluator could walk through an input structure more complicated than a tree, like a graph for instance.

References

[1] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory*, 2, no. 2, pp. 127–145, June 1968, Correction: *Math. Systems Theory*, 5, no. 1, pp. 95–96, Mar. 1971.

[2] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.*, 13, no. 3, pp. 229–256, 1980.

[3] J. Engelfriet and G. Filè, "Simple Multi-Visit Attribute Grammars," *J. Comput. System Sci.*, 24, no. 3, pp. 283–314, June 1982.

[4] J. Engelfriet, "Attribute Grammars: Attribute Evaluation Methods," in *Methods and Tools for Compiler Construction*, B. Lorho, Ed. New York, NY: Cambridge Univ. Press, pp. 103–138, 1984.

[5] D. Parigot, "Mise en œuvre des grammaires attribuées: transformation, évaluation incrémentale, optimisations," Univ. de Paris-Sud, Orsay, thèse de 3ème cycle, Sept. 1987.

[6] M. Jourdan and D. Parigot, "The FNC-2 System: Advances in Attribute Grammar Technology," INRIA, Rocquencourt, rapport RR-834, Apr. 1988.

[7] U. Kastens, "Lifetime Analysis for Attributes," *Acta Inform.*, 24, no. 6, pp. 633–652, Nov. 1987.

[8] J. Engelfriet and W. de Jong, "Attribute Storage Optimization by Stacks," Vakgroep Informatica, Rijksuniv. te Leiden, rapport 88-30, Dec. 1988, To be published.

[9] C. Julié, "Optimisation de l'espace mémoire pour l'évaluation des grammaires attribuées," Dépt. d'Informatique, Univ. d'Orléans, thèse, Sept. 1989.

[10] M. Jourdan and D. Parigot, *The FNC-2 System User's Guide and Reference Manual.* Rocquencourt, INRIA, Feb. 1989, This manual is periodically updated.

[11] U. Kastens, "The GAG-System—A Tool for Compiler Construction," in *Methods and Tools for Compiler Construction*, B. Lorho, Ed. New York, NY: Cambridge Univ. Press, pp. 165–182, 1984.

[12] R. Farrow and D. M. Yellin, "A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators," *Acta Inform.*, 23, no. 4, pp. 393–427, 1986.

[13] P. Deransart, M. Jourdan and B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography* (Lect. Notes in Comp. Sci.), vol. 323. New York–Heidelberg–Berlin, Springer-Verlag, Aug. 1988.